REMARKS

Claims 1-24 have been rejected as obvious over the combination of Goodwin et al. (US 6,199,195) in view of Features of VDM Tools and further in view of Koob et al.

In response to this rejection, the claims have been amended to point out the key differences over the combination of Goodwin and VDM Tools and Koob. **The key differences over Goodwin and VDM Tools and Koob is that none of these references nor a combination thereof teach a tool which provides a user interface mechanism to solicit data from the user which defines primitives which define a functional model (valuations, i.e., mathematical or Boolean equations which specify the effect specified events have on the values of variable attributes of an object) or an object interaction diagram or to define trigger relationships or global transactions or state transition diagrams as part of a dynamic model. The functional model, object interaction diagram and dynamic model are very important parts of a conceptual model of a complete program.**

Neither Goodwin nor VDM Tools nor Koob teach a specification which includes a functional model, an object interaction diagram, a dynamic model or a state transition diagram. Because these elements are not present in the specification or model created by these prior art references, it is clear that no automatically generated code which implements the desired functionality embodied in the functional model, dynamic model trigger relationships, global transactions or state transitions will be generated by these prior art tools of Goodwin, VDM Tools or Koob.

For a complete list of primitives that can be specified by the tool of the claimed invention which cannot be specified in the prior art tools, see Appendix A attached.

What is different about our invention then is that the computer is programmed with a different kind of a CASE tool than prior art CASE tools because it provides tools by which the user can specify valuations, triggers, local and global transactions and presentation patterns all of which define a complete conceptual model. These element that the invention can specify which the prior art tools cannot specify (valuations, triggers, local and global transactions and presentation patterns) means that the prior art tools cannot be used to create a complete conceptual model of a desired computer program which can be automatically converted to working code without the need for any hand writing of code. These elements like the functional model, object interaction diagram, dynamic model and state transition diagram help define a complete system. The primitives are converted to formal language statements of a formal language specification which encodes the conceptual model. These formal language statements can be validated to ensure they are complete, correct and not ambiguous. The validated formal language specification is used to automatically generate bug free code in some claims so that no manual writing of code is necessary and working bug-free code can be generated in a fraction of the time it normally takes to write complex computer programs.

The prior art Goodwin tool has tools to generate an object model in a modelling language such as UML. UML is not a formal language like OASIS which is the formal language of the preferred species of the invention. UML is is more about graphical notation than about semantics. One of the major drawbacks of UML is that it lacks precisely defined semantics for every element in the graphical notation. That means that by definition, the specification generated from a graphical model created in UML will not be a formal language specification since some of the UML primitives in the graphical model have no defined counterpart in the specification.

In contrast, OASIS is an object-oriented formal language based upon mathematics and logic for which there is a defined counterpart for every primitive in the graphical conceptual model, and every primitive is defined to have one and only one meaning so no ambiguities can arise in using the primitives of OASIS. The CASE tool of the claimed invention includes graphical user interface tools which can be used by the user to enter primitives which are defined in OASIS and define a functional model, dynamic model and object interaction diagram. Another big difference is what a designer can "say" in the OASIS language which he or she cannot "say" in UML (the functional model and the presentation model).

Although the VDM Tools prior art uses the terminology "formal specification", this is not a formal language specification because the VDM Tools reference teaches that parts of the VDM-SL formal specification is "non-executable" (see the second paragraph of the section entitled the C++ Code Generator). By definition, this is not a formal language specification because all parts of a formal language specification are executable meaning they can be automatically converted to code.

The elements of the conceptual model which are not present in the prior art Goodwin, VDM and Koob et al. references and which the computer defines user interface tools to allow the user to define include: a functional model; an object interaction diagram; a dynamic model and and state transition diagrams. The presentation model is also not present These elements of the conceptual model are necessary to define a complete and operative system for automatic code generation including an interface although the presentation model can be deleted to automatically generate an operative core of a program for which a user interface can be written by hand.

To highlight the differences over the prior art, the following amendment was made to the first programming element:

a software-generating computer programmed to:

> present user interface mechanisms to allow a user to enter input and receiving and storing ~~receive~~ user input data that defines a plurality of primitives which together define a conceptual model which models a complete system for which a complete computer program is to be automatically written and displaying at least some of said primitives of said conceptual model graphically, said primitives of said conceptual model defining an object model which specifies the system class framework which specifies constant, variable and derived attributes, a set of services including private and shared events and local transactions and integrity constraints specified for the class, and derivation expressions corresponding to derived attributes, and a functional model which specifies dynamic formulates related to evaluations where the effect of events on variable attributes is specified, and a dynamic model which specifies a state transition diagram for each class which specifies service preconditions which are formulas labeling state transitions and a process definition of a class which specifies a template of valid object lives and an object interaction diagram which specifies trigger relationships and global transactions;

Suppport for these amendments to claim 1 come from the following passage from page 39 et. seq. of the specification (emphasis in original).

> After having presented Conceptual Model and the OASIS formal concepts associated with them in accordance with one embodiment of the present invention, the mappings will now be discussed that generate a textual system representation 215 (that is a specification in OASIS) taking as input the graphical information introduced in the Conceptual Model. This formal specification 215 has in fact been obtained using CASE tool 210, and constitutes a solid system documentation to obtain a final software product which is compliant with the initial requirements, as represented in the source Conceptual Model.
>
> According to the class template introduced in the previous section,

the set of conceptual patterns and their corresponding OASIS representation.

The system classes are obtained from the object model. For each class, there are a set of constant, variable or derived attributes; a set of services, including private and shared events and local transactions; integrity constraints specified for the class; and derivation expressions corresponding to the derived attributes. For a complex class (those defined by using the provided aggregation and inheritance class operators), the object model also provides the particular characteristics specified for the corresponding complex aggregated or specialized class.

The information given by the object model basically specifies the system class framework, where the class signature is precisely declared. The dynamic model uses two kind of diagrams, the state transition diagram and the object interaction diagram. From the state transition diagram, the following are obtained: event preconditions, which are those formulas labeling the event transitions; the process definition of a class, where the template for valid object lives is fixed. From the object interaction diagram, two other features of an OASIS class specification are completed: trigger relationships and global transactions, which are those involving different objects.

Finally, the functional model yields the dynamic formulas related to evaluations, where the effect of events on attributes is specified. Having thus clearly defined the set of relevant information that can be introduced in a Conceptual Model in accordance with an embodiment of the present invention, the formal specification 215 corresponding to the requirements 200 provides a precise system repository where the system description is completely captured, according to the OASIS object-oriented model. This enables the implementation process (execution model) to be undertaken from a well-defined starting point, where the pieces of information involved are meaningful because they come from a finite catalogue of conceptual modeling patterns, which, furthermore, have a formal counterpart in OASIS.

Neither Goodwin nor VDM tools provide a tool which allows the user to define a conceptual model including a functional model and a dynamic model. These elements are necessary to define the functional part of an operative computer program which can have its code automatically generated.

Goodwin only teaches a tool by which the user can define an object model. An object model defines the classes, i.e., the static aspects of an object-oriented application, but not its dynamic nor functional aspects. Goodwin does not teach any graphicaly or other user interface tools a designer of a computer program can use to enter data to define a functional model or a dynamic model including triggers and/or global transactions and a state transition diagram (triggers, global transactions and state transition diagrams are all part of the dynamic model). Functional models and dynamic models are not part of the specification that Goodwin users define. Goodwin's tool can only be used to define an object model and not the "methods" or internal functionality of services provided by the objects the Goodwin tool can be used to define. The code that implements the internal functionality of each service must be written by hand, and the Goodwin tool does not provide any user interface mechanisms to model the functionality of the services defined using the tool.

This lack of user interface tool to define functional models and dynamic models is a crucial difference between Goodwin and the invention at bar because it means that the model created by the Goodwin tool does not correspond to the functional parts of a complete system and cannot be translated into the functional parts of a complete computer program which includes all the elements needed to do the intended job (other than the user interface --the presentation model is intentionally left out of claim 1 because a conceptual model without it still defines the functional part of a complete and operative program for which a user may wish to define a custom user interface without using the pattern primitives of the presentation model).

A designer using the Goodwin tool must still hand write the code that implements the internal functionality of the services each object performs (see the next paragrah for proof). Likewise, we could find no evidence that the object templates are automatically generated. Since Goodwin teaches that the object templates are inputs to the code generator, it is fair to say that they are probably not code which is automatically generated by the code generator and must be manually written.

The way functionality is expressed in Goodwin is by means of "routines". As taught in Col. 4, Lines 1-3, "An 'object class' is a set of data (attributes) and functional capabilities (routines) encapsulated into a single logical entity". At Col. 4, Lines 60-61, "A 'routine' is a functional capability associated with an object class". Routines are also referred to in the Goodwin application as "methods" " (see Col. 7, lines 23-28: "Large complex systems are composed of hundreds, if not thousands of business objects, and, various service specifications, such as CORBA common object service specifications (i.e., security, transaction, life-cycle, proxy, etc.), and require methods and behaviors to be implemented on each business object.")

Evidence that Goodwin does not automatically generate the code of the "methods" or services of the objects in the object model is found at Col 17,

Lines 2-14. There Goodwin teaches:

> The second step is to run the code generator 330 to generate the source code objects that will support the services in the Interface Definition Language (IDL) file that describes the interface of the objects to the business applications. (This is also a development time step.) The third step consists of compiling the Interface Definition Language (IDL) file to generate the sub code to connect the client 338 and the server. (This is a development time step.) **The fourth step is adding the code from the developer that actually implements the methods defined for the unified models, and any custom services or user defined code from the developer**. (Business logic is included in this step; this step is a development time step.). The fifth step consists of compiling all of the sources of the project, which are stub code generated by the interface definition language (IDL) compiler from the interface definition language (IDL) files generated by the code generator 330 (this code is used to connect the client 330 and the server in a CORBA system), the code generated by the code generator 330 to support the services; **the developer's code, which contains the business logic; and any library used by the developer to implement the business logic** or the stub code or the code generation code. (This is a development time step.) The sixth step consists of installing the "executable" code in a directory or store where the data server 332 can retrieve it in order to instantiate the objects. (This is also a development time step.)

This passage's bold parts clearly indicate that the code that actually implements the methods of the unified model (the object's services) which define the business logic of the system (where the rubber meets the road) is hand written code and is not automatically generated. While some of the business logic may be implemented using library code, that code is not automatically written by the code generator.

The applicants wish to make it clear that they are not claiming a system which can automatically generate code of a complete application. They already have a patent on that. The intent of this claim set is to claim the front end system which provides tools that allow a user to define a conceptual model including an object model, a functional model, a dynamic model and other things detailed in the amended claim 1 and the other amended claims. The claimed tool then converts this complete conceptual model of a complete system into a formal language specification written in a formal language such as OASIS which is based upon logic and mathematics such that it has precise rules of syntax and semantics such that the statements in the specification can be validated to ensure the formal language specification is complete, correct and not ambiguous to generate a validated formal language specification. If errors have been made in constructing the conceptual model, information is displayed which prompts a user to correct the primitive he entered to define the conceptual model so as to remove the errors.

It is the validated formal language specification which is converted by translators into a complete working computer program which, because it has been validated, will be bug free. The translation process is not part of the broadest claims.

The VDM Tools prior art also does not teach a tool which provides user interface mechanisms by which a user can define a functional model or a dynamic model comprising an object interaction diagram to define trigger relationships or global transactions and state transition diagrams. Neither does the Koob reference teach a tool which provides user interface mechanisms by which a user can define a functional model, an object interaction diagram to define trigger relationships or global transactions, a dynamic model or state transition diagrams.

Further Arguments Addressing Points Raised By The Examiner

First, the Examiner raised the issue about automation of generation of code and pointed out that VDM tools automatically generates some code (as does Goodwin). The invention is not about automatically generating just any code. It is about how much more program code can be automatically generated using the invention than when using the prior art tools. Using the tool of the invention, a conceptual model including an object model, functional model, dynamic model and presentation model can be created using the user interface tools available to enter and store primitives. Those primitives are automatically converted to statements in a formal language with predetermined rules of syntax and semantics. Those statements are then validated to make sure they are complete, correct and not ambiguous. If any errors have been made, information to that effect is brought to the attention of the user who can then correct the primitives to correct the error. The result is a validated formal language specification which can be automatically converted to code which is complete and bug free and even includes a user interface. The prior art cannot created a validated formal language specification of a complete computer program. If the presentation model is not included, as is the case with some of the claims, the tool still distinguishes over the prior art because the prior art Goodwin, VDM tools and Koob cannot model a functional model or dynamic model so the code automatically generated by such a combination would not be as complete as the code generated by the claimed invention.

The validated "formal language specification" which includes a an object model, a functional model and a dynamic model is the key element the creation of which allows the automatic generation of bug free code which is much more complete than can be created in the prior art. The term "formal language specification" cannot be found in Goodwin nor any of the references cited by the examiner. The specifications taught by Goodwin are "object models" which are "a formal description of an object-oriented application. Semantic elements of an object model describe object classes, relationships between object classes and inheritance between object classes" (Col 1. lines 18-12).

It is the functionality encoded into the functional model and dynamic model which makes the invention so much more useful than the prior art tools. The way functionality is expressed in Goodwin specifications is by means of "routines". As taught in Col 4, lines 1-3: "An "object class" is a set of data (attributes) and functional capabilities (routines) encapsulated into a single logical entity." and in Col. 4, lines 60-61: "A "routine" is a functional capability associated with an object class."

Routines are also referred to in the Goodwin application as "methods" (see Col. 7, lines 23-28: "Large complex systems are composed of hundreds, if not thousands of business objects, and, various service specifications, such as CORBA common object service specifications (i.e., security, transaction, life-cycle, proxy, etc.), and require methods and behaviors to be implemented on each business object.")

No mention as to how "routines"/ "methods", *i.e. business logic or functionality,* are automatically generated is found in the Goodwin patent. However, Goodwin does indicate that the functionality is added "by the developer" as seen in Col. 17, Lines 2-24, where we find (as previously quoted):

> "The second step is to run the code generator 330 to generate the source code objects that will support the services in the Interface Definition Language (IDL) file that describes the interface of the objects to the business applications. (This is also a development time step.) The third step consists of compiling the Interface

Definition Language (IDL) file to generate the sub code to connect the client 338 and the server. (This is a development time step.) The fourth step is adding **the code from the developer that actually implements the methods defined for the unified models**, and any custom services or user defined code from the developer. (Business logic is included in this step; this step is a development time step.). The fifth step consists of compiling all of the sources of the project, which are stub code generated by the interface definition language (IDL) compiler from the interface definition language (IDL) files generated by the code generator 330 (this code is used to connect the client 330 and the server in a CORBA system), the code generated by the code generator 330 to support the services; **the developer's code, which contains the business logic**; and any library used by the developer to implement the business logic or the stub code or the code generation code. (This is a development time step.)"

<u>Goodwin clearly teaches in this passage that the methods and business logic or functionality is written by hand and is not modelled by the Goodwin tool nor are any user interface mechanisms to create such a model provided by the Goodwin tool.</u> Here it does not say that the fourth step is optional, nor that "the code from the developers that actually implements the methods defined for the unified models" is optional. Moreover, it is saying that "code from the developers ... actually implements the methods defined for the unified models", which means that the implementation of part of the system MUST be done by developers, not by the invention disclosed in Goodwin's application. In the "unified models" only the **definition** of methods is found, but the **implementation** is done by a developer, not by Goodwin's invention.

This is further observed in this sentence "the developer's code ... contains the business logic" in reference to the "fith step" (which again is not optional). This "fifth step" consists of combining "all the sources of the project" which comprise:

- "stub code generated by the interface definition language (IDL) compiler from the interface definition language (IDL) files generated by the code generator 330 (this code is used to connect the client 330 and the server in a CORBA system),"
- "the code generated by the code generator 330 to support the services"
- "the developer's code, which contains the business logic"
- **"any library used by the developer to implement the business logic or the stub code or the code generation code"**

In our invention, the "formal language specification" is automatically generated from a conceptual model comprising four models as is discussed elsewhere herein. While in Goowin's invention, the "object model" or "logical model" is converted into a plurality of "unified models" but not into a "formal language specification" (see Abstract "A method for generating source code objects has steps of generating a plurality of logical models using a plurality of modeling tools; translating each of the plurality of logical models into corresponding ones of a plurality of unified models...").

Further evidence can be found in Goodwin's col. 4 lines 21-29, which teach that "An "object model" is a set of object classes that together form a blueprint for building an object-oriented application. Each object class of an object model can have attributes, inheritances, and relationships. Object models may be in the form of "logical models" generated by particular modeling tools and employing particular modeling languages, or "unified models" generated by a repository adaptor tool (or the like) and employing a unified modeling language, such as Unified Modeling Language (UML)".

So, Goodwin teaches here that both "logical models" and "unified models" are expressed in some "modeling language" but it does not teach that they are expressed in a "formal language". This is a significant difference because only specifications expressed in a formal language can be validated as is present in every claim of this invention. It is the validation process and the use of formal language which allows bug free code to be automatically generated.

In our invention, the Conceptual Model (the specification) comprises not just a single model (object model) but four: the Object Model, the Dynamic Model, the Functional Model and the Presentation Model. The Conceptual Model is automatically transformed into a formal language specification which is then used to "automatically write a complete working program from the formal specification".

This can be seen in the text of our patent, for example, here:

> "The system of the example described in the detailed description below, both the front end and back end (translation) processing is implemented. The front end processing captures the system requirements graphically (e.g. through a graphical user interface), **converts the four models so created into a formal specification**, and validates a high level repository written in a formal language from the four models for correctness and completeness. In the back end processing, **a translator is provided to automatically generate a complete, robust software application based on the validated formal specification."**

Goodwin also teaches that the objects which are modelled and converted automatically to "source code objects" are integrated into an "extensible object framework". The Goodwin reference does not even teach that the object services are automatically written. At col. 6, lines 29-36, Goodwin teaches:

> "The disclosed system and method allow object developers **(human programmers)** to design and author **(write by hand)** new object services, and to define how those services are composed **(hand written code to integrate the hand written services into the extensible object framework)** within extensible framework with other object services"

Parenthetical comments ours.

This means that a human must "author", i.e., write by hand, the object services and figure out how to integrate them into the extensible framework with other services. None of this is automatic code generation and none of it is specified in the model that can be created using the Goodwin system so no GUI tools to allow a user to enter primitives to specify a a functional model and dynamic model to define how the services affect the values of attributes and how the objects interact with each other (which would require a functional model and dynamic model) are present in Goodwin.

"Automatically generating source code" does not mean "automatically generating ALL the source code". Goodwin teaches that his invention "relates to automatically generated source code objects within extensible object frameworks"

A definition of "framework" is provided below

(Source: http://en.wikipedia.org/wiki/Framework)

> "In software development, a **Framework** is a defined support structure in which another software project can be organised and developed. Typically, a framework may include support programs, code libraries and a scripting language amongst other software to help develop and *glue together* the different components of your project."

A definition of "object framework" is provided below:

(Source http://jmvidal.cse.sc.edu/talks/corba/allslides.xml)

> "An object framework is a domain-specific group of objects that interact to provide a customizable solution within that application domain."

The "source code objects" automatically produced by Goodwin's invention do not account for a software system which is even close to as complete as provided by the claimed invention. This is because they need to be "within an extensible object framework" which will "glue them together" into an application. Said "extensible object framework" is NOT produced nor generated by Goodwin's invention. Therefore, what Goodwin's invention automatically produces is not as complete of a software system as what is specified in the validated formal language specification generated by the tool of the claimed invention.

Further evidence of this is found in Col. 6, lines 51-58 where Goodwin teaches the use of a given "extensible object framework" (namely CORBA) as one embodiment of his invention:

> "Executable programs (source code objects) are written by the code generator 210 in, for example, platform independent Java programming language, using the Common Object Request Broker Architecture (CORBA) 2.2 compliant, VISIBROKER Object Request Broker (ORB) for distributed objects. As a result, these source code objects are executable on any computer that has an installed Java browser and on which the VISIBROKER ORB exists."

Hence, source code objects are "executable programs" **only** when in conjunction with an ORB (Object Request Broker) or, more generically, as taught in Goodwin's invention, with "an extensible object framework". Goodwin, however does not teach that his invention automatically produces such ORB or, more generically, "extensible object framework". In fact, Goodwin teaches that the "extensible object framework" is hand written code at Col. 6, ines 37-41:

> The system and method will also allow developers **(human programmers)** to generate **(automatic code generation)** based on a framework of services they author **(they write by hand)** by composing services **(write service code by hand)** based on the object templates into objects that support the composed behaviors and methods.

Parenthetical comments ours.

Clearly then the services of the objects and the extensible object framework in which the automatically generated objects are integrated are hand written code which is not modelled nor validated by the tool of Goodwin.

The VDM Tools reference does not generate a formal language specification. The proof of this is as follows. It is true that nothing cited in "Features of VDM Tools" indicates that the specifications "must be" unstructured, only that they "can be" unstructured. **The point we want to make here is that a formal language specification "must be" structured (have a well-defined structure) which leaves no possibility for a formal language specification to be unstructured.** That proves the VDM tools specification cannot by definition be a formal language specification. If a given specification "can be" unstructured, which is the case of VDM Tools specifications, it is obvious that it is not a formal language specification, since a formal language specification "cannot be" unstructured.

The Examiner takes the position that the VDM Tools reference teaches validation of a formal language specification. Validation as taught in our specification and as claimed in our claims involves checking both syntax and semantics. The VDM Tools reference does not teach validation of semantics of their specification. VDM Tools automates syntax and type checks, which ensure the correctness of syntax and types, <u>but it does not teach automatic (nor even manual) checks for semantics.</u> In all, VDM Tools seem not to be concerned with completeness nor non-ambiguity, but just with correctness as can be seen in these two paragraphs (First paragraph of the section entitled "Syntax Checker"):

> "The Syntax Checher checks if the **syntax** of your specification is **correct**"

(First paragraph of the section entitled "Type Checker"):

> "When a file has been syntax checked it is possible to check the modules/classes in that file. This is a static check for **correctness** of the use of the VDM concepts in the specification, i.e. whether the operands are of the right type for the operators they have been used for and whether the variables are in scope:"

With regard to the claims that add a translator to automatically generate code, VDM Tools does not create full working program but only prototype code that needs to be hand modified and “fleshed out” to make a real program. It is only an outline or skeleton. Our tool creates a full working program including a user interface at least in claims that include a translator and a presentation model.

Our point here is to differentiate what is automatically generated by VDM Tools and what is automatically generated by our invention. The code which is produced by the code generator of VDM Tools is not that of a full application, but rather (and at best) a prototype. This can be seen in the following passage from the Second paragraph of the section entitled "Interpreter and Debugger":

> "One of the benefits of executing specifications is that testing techniques can be used to assist validation of the specifications. In the development process, small examples for parts of a specification can be executed to enhance the designer's knowledge of, and confidence in the specification. Furthermore, **an executable specification can form a running prototype.**"

The distinction between full applications and plain prototypes is in made in our invention from page 92, line 25 et seq.:

> "To sum up, the code automatically produced by the automatic software production system of one embodiment of the present invention corresponds to that of a true final software application, instead of that of just a prototype. To maintain this distinction, some of the differences between the generated system logic code from that of a prototype are explained.
>
> (1) Completeness: A prototype does not fully cover functionality of an information system, nor is it intended for every possible flow of execution, while our automatically generated code, being a final application, totally covers the functionality captured in the corresponding Conceptual Model, as well as every possible flow of execution.
>
> (2) Correctness: A prototype aims to verify user's needs and requirements and verify correctness of execution. The automatically generated code in accordance with an embodiment of the present invention, on the other hand, aims to verify user's needs and requirements, for it is correctly generated.
>
> (3) Robustness: A prototype is not robust, because the prototype is not produced with error checking and handling code. Rather, this code is not produced, typically by hand, until the very last step of codification, where user's needs and requisites have proven to be satisfied and a final application can then be produced. A final application, such is the case of our automatically generated code, must come with all the code necessary to assure robustness. Since this is usually codified by hand, programmers often forget to add such code in many places where needed. This leads to high costs of maintenance and disrupts the balance between system logic code and error checking and handling code. The system logic translators described herein provides all the necessary (and just than the

necessary) code to deal with error checking and handling.

> (4) Scalability: Prototypes are not scalable because they tend to be discarded during the process of validating user's needs and requisites. Final applications can be designed to be scalable because they aim to last much longer than a prototype. Nevertheless scalability implies following certain guidelines during design phase. With embodiments of the invention, system analysts need not worry about scalability because such a task falls under the System Logic Translator 232 responsibilities. So, analysts focus on analysis matters knowing that the resulting code will be scalable. Furthermore, different Conceptual Models translated by the System Logic Translator can interact with each other through their well-defined interfaces."

Another point is that the code generator of VDM Tools does not produce code for every part of the specification, as can be seen in the following passage from the second paragraph of the section entitled "The C++ Code Generator":

> "VDMTools generates fully executable code for **most** of the VDM-SL/VDM++ constructs leaving facilities for including user defined code for **non-executable parts of the specification**".

The fact that there are parts in a VDM Tools specification which are not executable means that even if a VDM Tools specification captured all the functional aspects of an information system (which obviously is not the case, since nothing in the reference claims that a specification captures all the semantics) not all parts of the specification would be automatically transformed into code. Hence, VDM Tools is not capable of producing full applications as is the tool of the claimed invention for claims that include translators and the presentation model.

Koob does not provide the missing elements needed to make the invention that are not present in VDM tools or Goodwin. Again, we do not say that Koob's system is not automatic, because it does automate the generation of "proof obligations". The distinction to make here is about what said "proof obligations" check. They do not check the specification itself but rather an implementation created from it. Koob et al. indicates that "proof obligations" are a set of "requirements **for ensuring the correctness of the implementation** with regard to the specification" (see first paragraph of page 4).

Further evidence that VSE does not check the specifications can be found in the abstract:

> "The proof obligations are automatically generated by the VSE system.

> The corresponding proofs can be carried out semi-automatically with VSE support. **This process guarantees the correctness of refinements**"

So, what is checked is NOT the specification itself but the implementation obtained from it or the refinements applied to the specification in order to obtain an implementation. Therefore, the last part of claim1 "validating said formal specification using the rules of syntax and semantics of said formal language, and verifying that every statement in said formal specification is syntactically complete, semantically correct and not ambiguous" is not anticipated by Koob et al.

Koob therefore does not teach modelling of a functional model and dynamic model nor does it teach validation of both the syntax and semantics of a formal language specification of the desired software program.

The Law Regarding Obvious Rejections Supported by Combinations of References

The CAFC has held that for the prima facie case obviousness to exist based upon a combination of references, the prior art itself must suggest to those skilled in the art that they should make the combination, and the prior art (and not the applicant's disclosure) must contain teachings that would lead one of ordinary skill in the art to have a reasonable expectation of success. In re Vaeck, 947 F.2d 488 [20 USPQ2d 1438] (Fed. Cir. 1991). Reasonable probability of success in solving the problem by making the combination must be apparent from the references themselves. Where the prior art does not contain all the knowledge needed to solve the problem, it is unlikely that the Federal Circuit would find that a reasonable expectation of success could be found in the prior art itself since not all the building blocks needed to make the invention are present in the prior art combination.

Both the suggestion and the reasonable expectation of success must be founded in the prior art, not in the applicant's disclosure. *See* In re Dow Chemical Co., 837 F.2d 469, 473, 5 U.S.P.Q.2D (BNA) 1529, 1531 (Fed. Cir. 1988).

Here, none of the prior art references in the combination teach a CASE tool which provides user interface mechanisms to define a Conceptual Model which includes:

> an object model which specifies the system class framework which specifies constant and variable and derived attributes, a set of services including private and shared events and local transactions and integrity

constraints specified for the class, and derivation expressions corresponding to derived attributes, and a functional model which specifies dynamic formulates related to evaluations where the effect of events on variable attributes is specified, and a dynamic model which specifies a state transition diagram for each class which specifies service preconditions which are formulas labeling state transitions and a process definition of a class which specifies a template of valid object lives and an object interaction diagram which specifies trigger relationships and global transactions;

In other words, neither UML nor VDM-SL includes elements of the language which can be used to express a functional model with dynamic formulas or a dynamic model to specify state transition diagrams and object interaction diagrams. Therefore, neither the Goodwin prior art tool nor the VDM Tools prior art tool include user interface mechanisms by which a designer can enter data defining the functional model or the dynamic model.

Examples of the user interface mechanisms provided in the invention to define these various elements of the conceptual model such as state transition diagrams, functional models, dynamic models follow.

An example of the user interface mechanism used by developers to create a state transition diagram and an actual state transition diagram example are shown in Figure 4A and described in the text starting at page 25. The menu choices and icons in the screen shot of Figure 4A provide the tools to defines states and transitions for various objects to define a complete life cycle for every object.

An example of the user interface mechanism to provide a tool to enter data defining an object interaction diagram is shown in Figure 4B, and the text which describes this part of the tool starts at page 26. An object interaction diagram specifies triggers that cause certain services to be executed to change the state of an attribute of an object or do something to an object. The trigger graphically illustrated at 424 indicates that a reader punish action is to be invoked to act on the reader object 340 in Figure 3 when the number of books a reader has borrowed reaches 10.

An example of the user interface mechanism to provide a tool to enter data defining a functional model is shown in Figure 5, and the text which describes this part of the tool begins on page 26. The functional model functions to record data entered by a user using the user interface tools of Figure 5 to define the semantics of how an object state will be changed as a result of an event occurrence. More clearly stated, as quoted from page 26 of the specification:

> Basically, the functional model allows a SOSY modeler to specify a class, an attribute of that class and an event of that class and then define a mathematical or logical formula that defines how the attribute's value will be changed when this event happens.

What is missing from the combination of prior art references applied to the claims is the "primitives" of the functional model, the dynamic model, and the presentation mode and the user interface tools which allow a user to create instances of these primitives in the conceptual model. These primitives include but are not limited to: valuation formulas; trigger relationships, global transactions; service preconditions; class name and the ability to specify an attribute of that class for a valuation formula to act upon; state transitions; etc. Because these primitives have no analogs in VDM-SL or UML, the tools that are used in the Goodwin and VDM Tools reference do not have functionality to allow a user to create instances of such primitives in the models built with those tools.

Therefore, the combination of prior art references lack critical elements needed to make the invention: there is no teaching of a computer programmed with a program that creates user interface mechanisms to create instantiations of these missing primitives; and there is no teaching of a program which can control a computer to present users with user interface mechanisms by which users can create instantiations of these missing primitives; and there is no teaching of a computer-readable medium upon which is stored such a program.

Where the prior art combination is missing a critical element of the invention, it is not fair to say the prior art combination suggests a a probability of success in solving the problem the invention solves by combining the references along the lines of the claimed invention. This is because the combination would still lack the critical element needed to make it work. Accordingly, claim 1, as amended, is not obvious from the combination of references applied against it.

Claim 2 depends from claim 1, and has been amended voluntarily to improve its form.

Claim 3 depends from claim 1, and has been amended voluntarily to specify that the system logic translator translates the object model, functional model and dynamic model into working code.

Claim 4 depends from claim 1, and has been amended to specify that the computer is programmed to provide tools to define a presentation model which models the desired user interface for the automatically generated computer program and to specify that the user interface translator processes the presentation model statements in the formal language specification to write code automatically which implements the desired user interface.

Claim 5 depends from claim 1 and has been amended to more clearly define that the database generator works on the object model statements in the formal language specification to automatically create a database which can store the values of attributes of each class defined in the object model. Support is found in the following passage from page 18 of the specification:

> ...the structure of the database or table or other data structure that database generator 236 creates is defined by the objects and classes defined in the Conceptual Model.

Claim 6 has be voluntarily amended to improve its form.

Claim 7 is an independent process claim, and has been amended in response to the obviousness rejection thereof to add that the process includes a step of:

> <u>displaying on a computer user interface mechanisms which can be used by a designer of said desired computer program to create instances of primitives defining a conceptual model of said desired computer program, said conceptual model comprised of an object model, a functional model, a dynamic model and a presentation model,</u>

The step of displaying user interface mechanisms that can be used by a designer to create instances of primitives of a functional model and a dynamic model is not present in the prior art combination of Goodwin, VDM Tools and Koob because neither the UML nor VDM-SL languages allow users to create statements therein which define either a functional model, a dynamic model or a presentation model. The languages used by the Goodwin and VDM Tools prior art are not a formal language like Oasis (which the invention uses in one specific embodiment within its genus). Neither UML nor VDM-SL allow a user to define functional models, dynamic models or presentation models. Therefore, these prior art processes will not include a step of displaying user interface tools which can be used to create instances of the primitives that are used to construct functional models, dynamic models or presentation models.

The following language was added to the validation step of claim 7:

> and if one or more errors are found, to display information to allow a user to adjust the primitives recorded as part of said conceptual model to correct said error(s).

This language makes it clear that the validation step does not automatically correct error it finds in the conceptual model. Instead, it displays information which prompts the user to make corrections in the primitives recorded as part of the conceptual model to correct the errors the validation step found.

It is, among other things, the step of displaying user interface mechanisms that can be used to create instances of the primitives that make up the functional model, dynamic model and presentation model which distinguishes the process of claim 7 over any process which can be gleaned from the prior art combination. The functional model, dynamic model and presentation model are key elements of a conceptual model of a complete and operable computer program which can be automatically generated.

The fact that the prior art Goodwin and VDM Tools and Koob references do not allow a user to define a functional model, dynamic model or presentation model means that any process derived from a combination of any of these references would be missing a key element of knowledge needed to make the invention of claim 7. Therefore, it is not fair to say that the prior art references Goodwin and VDM Tools and Koob would create in the mind of the skilled artisan a reasonable expectation of success in solving the problem the invention of claim 7 solved. This is because even if a combination of the teachings were to be made, the combination would still be lacking several key elements needed to make the claimed invention, and one skilled in the art would realize this. Since once skilled in the art would not perceive a liklihood of success, suggestion to make the combination does not exist and the combination cannot be used to support an obviousness rejection of any claim that has been amended along the lines that claim 1 has been amended.

Claims 8-14 depend from claim 7, and are now allowable for the same reasons as claim 7 since they contain claim 7 limitations that are not found in the prior art and which are not suggested thereby.

Claim 15 has been amended along the same lines as claim 7, and is not obvious from the prior art combination applied against it. This is because the amendment adds limitations regarding displaying user interface mechanism that can be used to define a functional model, a dynamic model and a presentation model, none of which can be defined in the prior art references.

Claim 16 depends from claim 15 and is not obvious for the same reasons claim 15 is not obvious. A few voluntary amendments have been made to the language of claim 16 to conform its terminology to the amended terminology of the parent claim.

Claim 17 has been voluntarily amended to improve its form and to specify that the step of receiving user input that defines instances of primitives that define said functional model. Since claim 17 claims a process for displaying user interface mechanisms through which data can be entered to define primitives of a functional model, claim 17 distinguishes over the prior art because the prior art combination applied against the claims does not include teachings of modelling of functional models as part of the modelling process. Therefore, the prior art does not include teachings of presentation of user interface mechanisms that can be used to enter data that define instances of primitives of a functional model of a desired computer program.

Claim 18 is similar to claim 17 and is not obvious for the same reasons claim 18 is not obvious. The claim defines a process to display user interface mechanisms for definining a functional model and then receiving user input through those mechanisms which define primitives that define the desired functional model.

Claim 19 is similar to claim 15 except that it defines a computer programmed to present user interface mechanisms by which a user can enter data defining primitives which define classes of objects and attributes of those classes and mathematical or logical expressions which define conditions and effects which act upon the values of variable attributes of said classes so as to define the object model and functional model (the functionality) of a desired computer program being modelled. Since the prior art applied against the claim does not teach modelling a functional model and the knowledge needed to make the claimed invention is not present in the prior art nor suggested by the prior art, claim 19 is not obvious. One skilled in the art would not perceive any liklihood of success in solving the problem the invention solves from making the combination the Examiner applied since even if the combination could be made technically, it would still be lacking an important element of knowledged needed to make the claimed invention: the ability to define a functional model which specified the desired functionality of the desired computer program.

Claim 20 is the same as claim 19 but elements of programming of the computer have been added to specify a computer programmed to automatically convert the primitives into formal language statements and to use the rules of syntax and semantics to validate the formal language statements to ensure they are complete, correct and not ambiguous, and to prompt the user to edit the primitives to correct any errors which were found.

Claim 21 has been amended in response to the prior art rejection to specify that the user interface mechanisms the computer is programmed to display allow the user to enter primitives that define an object model, a functional model and a dynamic model. That alone distinguishes the claim over the prior art since the prior art does not display user interface mechanisms to define a functional model or a dynamic model. No presentation model is included because it may be desirable to write this part of the computer program by hand to obtain more flexibility in the user interface. The claim has been further amended to define the programming of said computer to control it to receive user input defining instances of said primitives, automatically convert those instances into formal language statements in a mathematically based formal language which has precise, predetermined rules of syntax and semantics which are such that said rules can be used to validate the entire collection of formal language statements to make sure they define a complete, correct and not ambiguous specification of a computer program to be automatically written, the part being defined including specification of code which implements said object model, said functional model and said dynamic model and said presentation model. The prior art does not teach tools which allow a user to automatically convert primitives entered by a user into formal language specifications of a functional model and a dynamic model and to automatically validate said formal language statements and prompt a user to correct the primitives if errors are found.

Claim 22 depends from claim 21 and has been amended voluntarily to remove the original limitations which are now redundant and to substitute specification of programming of said computer which controls it to automatically translate said validated formal language specification into working computer code.

Claim 23 defines a computer readable medium which stores a data structure which contains data which define a functional model. This is not taught by the prior art because the prior art combination does not teach modelling a functional model or storing a data structure of data which defines the functional model primitives entered by the user.

Claim 24 has been amended in response to the prior art rejection to make it clear that the user interface mechanisms displayed allow a user to enter primitives which define a functional model and a dynamic model of the conceptual model of the desired computer program. The amendments also make it clear that these primitives are automatically converted to formal language statements which define a formal language specification which is validated using the precise, predetermined rules of syntax and semantics.

Respectfully submitted,

Dated: February 7, 2005

Ronald C. Fish

Ronald Craig Fish
Reg. No. 28,843
Tel 408 866 4777
FAX 408 866 4785

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail in an envelope addressed Commissioner for Patents , P.O. Box 1450, Alexandria, Va. 22313-1450 on
(Date Of Deposit) 2/7/05
Express Mail Receipt Number: EV 423 637 397 US

Ronald C. Fish

Ronald Craig Fish, President
Ronald Craig Fish a Law Corporation
Reg. No. 28,843

Appendix A – Distinctive Primitives Which Can Be Specified in a Conceptual Model Using the Tool of the Claimed Invention Which Cannot Be Specified By The Prior Art Tools

Local Transactions (Object Model)

Purpose

A transaction is a molecular service: that is, a service whose functionality is defined by a sequence of other services. The services composing a local transaction can be atomic (events) or molecular (transactions).
The combination of the valuations defined in the Functional Model (which determine the functionality of events) with the composition mechanism of transactions result in the ability to completely define the functionality of services (either atomic or molecular).

Supporting Figure(s) in our specification

Figure 13 **IN CHG-001.1P and CHG-001.2P CIPs**

Supporting Passages in our specification

Page 63, line 5
"FIG. 13 is a screenshot of the dialog box used to create one formula in a local transaction carried out by a composed service (single services are called events, and composed services are called local transactions)."

Page 20, line 3
"Services can be of two types: events and transactions. Events are atomic operations while transactions are composed of services which can be in turn events or transactions. Every service can have the following characteristics: name, type of service (event or transaction), service alias, remarks and a help message. Events can be of three types: new, destroy or none of them. Events can also be shared by several classes of the project. Shared events belong to all classes sharing them. Transactions have a formula that expresses the composition of services."

Page 27, line 3
"An "event" as used in the claims means a single service and not a transaction which is defined as a composed or complex service (which means more than one service executes)."

Page 39, line 1
"The molecular units that are the transactions have two main properties. First, they follow an all-or-nothing policy with respect to the execution of the involved events: when

a failure happens during a transaction execution, the resultant state will be the initial one. Second, they exhibit the non-observability of intermediate states."

Valuations (Functional Model)

Purpose

The functional model defines how the occurrence of events of a class change the values of attributes of said class. This is done by means of valuations.

Figure(s) in our specification

Figure 5
Figure 15 **IN CHG-001.1P and CHG-001.2P CIPs**

Passages in our specification

Page 12, line 12
"FIG. 5 illustrates an exemplary dialog for receiving input for the functional model."

Page 13, line 10
"FIG. 15 is a dialog box to enter the functional model formulas that define evaluation of the attribute "cause" with the "modify" event (an event is a single service). The functional model relates services mathematically through well-formed formulas to the values of attributes these services act upon."

Page 17, line 13
"In one implementation, the Conceptual Model is subdivided into four complementary models: an object model, a dynamic model, a functional model, and a presentation model."

Page 18, line 19
"Rather, four complementary models, that of the object model, the dynamic model, the functional model and the presentation model, are employed to allow a designer to specify the system requirements."

Page 19, line 7
"In accordance with the widely accepted object oriented conceptual modeling principles, the Conceptual Model is subdivided into an object model, a dynamic model, and a functional model. These three models, however, are insufficient by themselves to specific a complete application, because a complete application also requires a user interface."

Page 26, line 29
"One embodiment of the present invention, however, employs a functional model that is

quite different with respect to these conventional approaches. In this functional model, the semantics associated with any change of an object state is captured as a consequence of an event occurrence. To do this, the following information is declaratively specified: how every event changes the object state depending on the arguments of the involved event, and the object's current state. This is called. "valuation."

In particular, the functional model employs the concept of the categorization of valuations. Three types of valuations are defined.:push-pop, state-independent and discrete-domain based. Each type fixes the pattern of information required to define its functionality.
Push-pop valuations are those whose relevant events increase or decrease the value of the attribute by a given quantity, or reset the attribute to a certain value.
State-independent valuations give a new value to the attribute involved independently of the previous attribute's value.
Discrete-domain valuations give a value to the attributes from a limited domain based on the attribute's previous value. The different values of this domain model the valid situations that are possible for the attribute."

Page 26, line 29
"One embodiment of the present invention, however, employs a functional model that is quite different with respect to these conventional approaches. In this functional model, the semantics associated with any change of an object state is captured as a consequence of an event occurrence. Basically, the functional model allows a SOSY modeler to specify a class, an attribute of that class and an event of that class and then define a mathematical or logical formula that defines how the attribute's value will be changed when this event happens. An "event" as used in the claims means a single service and not a transaction which is defined as a composed or complex service (which means more than one service executes). In the preferred embodiment, condition-action pair is specified for each valuation. The condition is a single math or logic formula is specified which specifies a condition which results in a value or logical value which can be mapped to only one of two possible values: true or false. The action is a single math or logical formula which specifies how the value of the attribute is changed if the service is executed and the condition is true. In other embodiments, only a single formula that specifies the change to the attribute if the service is executed is required.

The functional model is built in the preferred embodiment by presenting a dialog box that allows the user to choose a class, an attribute of that class and a service of that class and then fill in one or more formula or logical expressions (condition-action or only action) which controls how the value of that attribute will be changed when the service is executed. The important thing about this is that the user be allowed to specify the mathematical or logical operation which will be performed to change the value of the attribute when the service is executed, and it is not critical how the user interface is implemented. Any means to allow a user to specify the class, the attribute of that class and the service of that class and then fill in a mathematical or logical expression which controls what happens to the specified attribute when the service is executed will suffice to practice the invention. Every one of these mathematical expressions is referred to as a valuation. Every valuation has to have a condition and action pair in the preferred embodiment, but in other species, only an action need be specified. The condition can be any well formed formula resulting in a Boolean value which can be mapped to only one of two possible conditions: true or false. The action specified in the pair is any other well-formed mathematical and/or logical formula resulting in a new value for the variable attribute, said new value being of the attribute's same data type (type of data of action must be compatible with the type of data of the attribute). This valuation formula can be only mathematical or only a Boolean logical expression or a combination of both mathematical operators and Boolean logical expressions.

Regardless of the user interface used to gather data from the user to define the valuations in the functional model, all species within the genus of the invention of generating functional models will generate a data structure having the following content: data defining the valuation formula which affects the value of each variable attribute (the data that defines the valuation formula identifies the service and the attribute affected and the mathematical and/or logical operations to be performed and any operands needed). This data structure can be any format, but it must contain at least the above identified content.

To define the functional model, the following information is declaratively specified by the SOSY modeler: how every event changes the object state depending on the arguments of the involved event, and the object's current state. This is called "valuation".

In particular, the functional model employs the concept of the categorization of valuations. Three types of valuations are defined.:push-pop, state-independent and discrete-domain based. Each type fixes the pattern of information required to define its functionality.

Push-pop valuations are those whose relevant events increase or decrease the value of the attribute by a given quantity, or reset the attribute to a certain value.

State-independent valuations give a new value to the attribute involved independently of the previous attribute's value.

Discrete-domain valuations give a value to the attributes from a limited domain based on the attribute's previous value. The different values of this domain model the valid situations that are possible for the attribute."

Page 29, line 1

"This categorization of the valuations is a contribution of one aspect of the present invention that allows a complete formal specification to be generated in an automated way, completely capturing a event's functionality.
Accordingly, the functional model is responsible for capturing the semantics of every change of state for the attributes of a class. It has no graphical diagram. Textual information is collected through an interactive dialog that fills the corresponding part of the Information Structures explained before. FIG. 5 illustrates an exemplary dialog for receiving input for the functional model."

Page 37, line 12

"Evaluations are formulas of the form _[a] _ whose semantics is given by defining a ρ function that, from a ground action a returns a function between possible worlds. In other words, being a possible world for an object any valid state, the ρ function determines which transitions between object states are valid after the execution of an action a. In the example,there are the following evaluations:

[loan( )] book_count=book_count+1;

[returns( )] book_count=book_count-1;

Within this dynamic logic environment, the formula Φ is evaluated in $s \in W$, and Φ' is evaluated in $\rho(a)$, with $\rho(a)$ being the world represented by the object state after the execution in s of the action considered."

Page 40, line 12

"Finally, the functional model yields the dynamic formulas related to evaluations, where the effect of events on attributes is specified."

Page 63, line 16

"Phase 4: Express evaluations. During this phase, one or more dialog boxes are presented to the SOSY modeler wherein he or she expresses evaluations of what will be the effect of all event for each variable attributes of each class.

This is the process of building the functional model portion of the Conceptual Model. The value change of an attribute when an event happens is known as "evaluation".

FIG. 15 is a dialog box to enter the functional model formulas that define evaluation of the attribute "cause" with the "modify" event (an event is a single service). The functional model relates services mathematically through well-formed formulas to the values of attributes these services act upon. Note that at box 724, the SOSY modeler has not filled in an evaluation formula that could be encoded in the final code to do a calculation to change the value of "cause" when the modify event occurs. Instead, as seen from box 726, the value of "cause" will be changed to whatever the value of the argument "p_cause" of the event "modify" when "modify" is executed."

Dynamic Model

Purpose

The dynamic model specifies the behaviour of an object in response to services, triggers and global transactions. It encompasses:

- The State Transition Diagram
- The Object Interaction Diagram

Figure(s) in our specification

Figure 4A (State Transition Diagram)
Figure 4B (Object Interaction Diagram)
Figure 17 (State Transition Diagram) **IN CHG-001.1P and CHG-001.2P CIPs**

Passages in our specification

Page 13, line 17
"FIG. 17 is one of the two graphical user interface diagrams of the dynamic model on which the SOSY modeler has drawn a graphic illustrating the state transitions for the "expense" class."

Page 24, line 20
"The dynamic model specifies the behavior of an object in response to services, triggers and global transactions. In one embodiment, the dynamic model is represented by two diagrams, a state transition diagram and an object interaction diagram.
The state transition diagram (STD) is used to describe correct behavior by establishing valid object life cycles for every class. A valid life refers to an appropriate sequence of states that characterizes the correct behavior of the objects that belong to a specific class. Transitions represent valid changes of state. A transition has an action and, optionally, a

control condition or guard. An action is composed of a service plus a subset of its valid agents defined in the Object Model. If all of them are marked, the transition is labeled with an asterisk (*). Control conditions are well formed formulas defined on object attributes and/or service arguments to avoid the possible non-determinism for a given action. Actions might have one precondition that must be satisfied in order to accept its execution. A blank circle represents the state previous to existence of the object. Transitions that have this state as source must be composed of creation actions. Similarly, a bull's eye represent the state after destruction of the object. Transitions having this state as destination must be composed of destruction actions. Intermediate states are represented by circles labeled with an state name.

Accordingly, the state transition diagram shows a graphical representation of the various states of an object and transitions between the states."

Page 25, line 15
"Transitions are represented by solid arrows from a source state to a destination state. The middle of the transition arrow is labeled with a text displaying the action, precondition and guards (if proceeds)."

Page 25, line 24
"The object interaction diagram specifies interobject communication. Two basic interactions are defined: triggers, which are object services that are automatically activated when a pre-specified condition is satisfied, and global transactions, which are themselves services involving services of different objects and or other global transactions. There is one state transition diagram for every class, but only one object interaction diagram for the whole Conceptual Model, where the previous interactions will be graphically specified.
In one embodiment, boxes labeled with an underlined name represent class objects.Trigger specifications follow this syntax:destination::acti- on if trigger-condition. The first component of the trigger is the destination, i.e., the object(s) to which the triggered service is addressed. The trigger destination can be the same object where the condition is satisfied (i.e. self), a specific object, or an entire class population if broadcasting the service. Finally, the triggered service and its corresponding triggering relationship are declared. Global Transactions are graphically specified by connecting the actions involved in the declared interaction. These actions are represented as solid lines linking the objects (boxes) that provide them.
Accordingly, communication between objects and activity rules are described in the object interaction diagram, which presents graphical boxes, graphical triggers, and graphical interactions. FIG. 4B illustrates an exemplary object interaction diagram 420 in

accordance with one embodiment of the present invention.
In the object interaction diagram 420, the graphical interactions is represented by lines for the components of a graphical interaction. Graphical boxes, such as reader box 422, are declared, in this case, as special boxes that can reference objects (particular or generic) such as a reader. Graphical triggers are depicted using solid lines that have a text displaying the service to execute and the triggering condition. Components of graphical interactions also use solid lines. Each one has a text displaying a number of the interaction, and the action that will be executed. In the example, trigger 424 indicates that the reader punish action is to be invoke invoked when the number of books that a reader is currently borrowing reaches 10."

Page 40, line 5
"The dynamic model uses two kind of diagrams, the state transition diagram and the object interaction diagram. From the state transition diagram, the following are obtained: event preconditions, which are those formulas labeling the event transitions: the process definition of a class, where the template for valid object lives is fixed. From the object interaction diagram, two other features of an OASIS class specification are completed: trigger relationships and global transactions, which are those involving different objects."

Page 64, line 17
"FIG. 17 is one of the two graphical user interface diagrams of the dynamic model on which the SOSY modeler has drawn a graphic illustrating the state transitions for the "expense" class. Each state in the state transition diagram represents a valid state for the object and represents one of the "valid lives" and really is one of the unseen attributes of the expense class. An object can only enter one of the displayed states if the corresponding service has been thrown to transition to it from a previous state."

[Col. 9, lines 28-32]
"A class can also store triggers. Each trigger may be composed of trigger target specified in terms of self, class or object, trigger condition, triggered action (service plus a list of possible agents) to be activated and a list of default values associated with the arguments of the related service."

Page 38, line 8
"Triggers are formulas of the form _[_a]false,, where ¬ a is the action negation. This formula means that a does not occur, and what does occur is not specified. If Φ holds and an action other than a occurs, then there is no successor state. This forces a to occur or the system remains in a blocked state. For instance, using the appropriate dynamic formula where we include in the triggered service information about the destination

(according to the trigger expressiveness presented when the object interaction diagram 420 was introduced), we will declare:

book_count=10 [Self::punish( )] false

This trigger may be written in an equivalent but more conventional way for specification purposes as:

Self::punish( ) if book_count=10;

Thus, triggers are actions activated when the condition stated in Φ holds. The main difference between preconditions and triggers comes from the fact that in triggers there is an obligation to activate an action as soon as the given condition is satisfied. In this way triggers allow us to introduce internal activity in the Object Society that is being modeled."

Presentation Model

Purpose

Describe user interface requisites in an abstract way, regardless of how these requisites are to be implemented.

Figure(s) in our specification

FIG. 19 (Display Set) **IN CHG-001.1P and CHG-001.2P CIP**
FIG. 20 (Filter) **IN CHG-001.1P and CHG-001.2P CIP**

Passages in our specification

Page 13, line 20
"FIG. 18 is a dialog box used by the SOSY modeler to establish this precondition."

Page 13, line 21
"FIG. 19 is a dialog box used by the SOSY modeler to establish the set of attributes which will be displayed for the "expense" class."

Page 11, line 16
"Another aspect of the present invention stems from the realization that a major source of inadequacy of conventional prototyping techniques is that these techniques lack the capability to specify the user interface aspects. Thus, such conventional prototypes have primitive user interfaces that are unacceptable for final, customer-ready software application. Accordingly, this aspect of the invention relates to an automated software production tool, software, and methodology that includes a formal specification of a

Conceptual Model that specifies requirements for a software application. The Conceptual Model includes a presentation model that specifies patterns for a user interface of the software application. The formal specification, which also specifies the presentation model is validated; and the software application is then generated based on the validated formal specification. As a result, the generated software application includes instructions for handling the user interface in accordance with the patterns specified in the presentation model."

Page 29, line 11
"The presentation model is a set of pre-defined concepts that can be used to describe user interface requisites. These concepts arise from distilling and abstracting repetitive scenarios in developing the user interfaces. These abstractions of the repetitive scenarios are called patterns. A set of patterns is called a pattern language.

In this sense, the presentation model is a collection of patterns designed to reflect user interfaces requirements. A pattern is a clear description of a recurrent problem with a recurrent solution in a given restricted domain and giving an initial context. The documented patterns abstract the essence of the problem and the essence of the solution and therefore can be applied several times to resolve problems that match with the initial context and domain. The pattern language is composed of a plurality of patterns. The present invention is not limited to any particular list of patterns, but the following is a brief description of some user interface patterns that have been found to be useful: Service presentation pattern, Instance presentation pattern, class population presentation pattern, master-detail presentation pattern and action Selection presentation pattern."

Primitive: Service Presentation Pattern
(currently referred to as Service Interaction Unit)
Page 29, line 25
"A service presentation pattern captures how a service will enquire data to the final user. This patterns controls the filling out of service arguments and contains actions to launch the service or to exit performing no action. It is based on other lower level patterns that refer to more specific interface tasks such as an introduction pattern, defined selection pattern, population selection pattern, dependency pattern, status recovery pattern, supplementary information pattern and argument grouping presentation"

Primitive: Introduction Pattern
Page 29, line 29
"The introduction pattern that handles with restrictions to input data that must be provided to the system by the final user (i.e., the user who employs the final

application). In particular, edit-masks and range-values are introduced, constraining the values that can validly be input in the interface. In this manner, the user-entry errors are reduced. This pattern can be applied to arguments in services or to attributes in classes to improve data input process through validating input arguments."

Primitive: Defined Selection Pattern
Page 30, line 3
"The defined selection pattern that specifies a set of valid values for an argument. When the input data items are static, are a few, and are well known, the designer can declare by enumeration a set containing such valid values. This pattern is similar to those that define an enumerated type and an optional default value. Accordingly, the final user can only select an entry from the pre-specified set, thereby reducing error prone input. For example, one representation of this pattern could be a Combo-Box. This pattern can be applied to arguments in services or to attributes in classes to improve data input process."

Primitive: Population Selection Pattern
(currently part of the Population Interaction Unit)
Page 30, line 11
"The population selection pattern that handles the display and selection of objects in a multiple objects society. Specifically, this pattern contains a filter, a display set, and an order criterion, which respectively determine how objects are filtered (Filter Expression), what data is displayed (Display Set), and how objects are ordered (Order Criteria). This pattern may be thought of as a SQL Select statement with columns, where for the filter expression and order by for the ordering clauses, and can be applied to object-valuated arguments in services whenever it is possible to select an object from a given population of existing objects."

Primitive: Dependency Pattern
Page 30, line 19
"The dependency pattern, that is a set of Event-Condition-Action (ECA) rules allowing the specification of dependency rules between arguments in services. When arguments are dependent on others, these constraints use this kind of rules."

Primitive: Status Recovery Pattern
Page 84, line 17
"The status recovery pattern is an implicitly created pattern that recovers data from object attributes to initialize service arguments. This can be modeled as an implicit set of dependency patterns."

Primitive: Supplementary Information Pattern
[Col. 15, lines 56-59]
"The supplementary information pattern handles the feedback that is provided to final users in order to assure they choose or input the correct OID (object identifier) for an existent object."
[Col. 15, lines 63-64]
"The supplementary information pattern is applicable to object-valuated arguments."

Primitive: Argument Grouping Presentation Pattern
[Col. 15, lines 65-67]
"The argument grouping presentation pattern, that captures how to group the requested service arguments according to the user wishes."

Primitive: Instance Presentation Pattern
(currently referred to as Instance Interaction Unit)
[Col. 16, lines 1-5]
"An instance presentation pattern captures how the properties of an object are presented to the final user. In this context, the user will be able to launch services or to navigate to other related objects. The instance presentation pattern is a detailed view of an instance."

Primitive: Class Population Presentation Pattern
(currently referred to as Population Interaction Unit)
[Col. 16, lines 6-10]
"A class population presentation pattern captures how the properties of multiple objects of one class are presented to the final user. In this context, once an object is selected, the final user will be able to launch a service or to navigate to other related objects. The objects can also be filtered."

Primitive: Master-Detail Presentation Patter
(currently referred to as Master-Detail Interaction Unit)
[Col. 16, lines 11-20]
"A master-detail presentation pattern captures how to present a certain object of a class with other related objects that may complete the full detail of the object. To build this pattern the following patterns are used: instance presentation, class population presentation and, recursively, master-detail presentation. In this manner, multi-detail (multiple details) and multi-level master-detail (multiple levels recursively) can be modeled. For example, one scenario involves an invoice header followed by a set of

invoice lines related to the invoice."

Primitive: Action Selection Pattern
[Col. 16, lines 21-28]
"An action selection pattern captures how the services are offered to final users following the principle of gradual approach. This pattern allows, for example, generating menus of application using a tree structure. The final tree structure will be obtained from the set of services specified in the classes of the Conceptual Model. The user could launch services or queries (observations) defined in the Conceptual Model."

Primitive: Filter Expression
[Col. 16, lines 29-40]
"A Filter Expression is a well-formed formula that evaluates to a Boolean type. This formula is interpreted as follows: the objects that satisfy the formula pass the filter; the ones that do not fulfill the condition do not pass the filter. Consequently, the filter acts like a sift that only allows objects that fulfill the formula to pass. These formulas can contain parameters that are resolved at execution time, providing values for the variables or asking them directly to the final user. A filter pattern may be thought of as an abstraction of a SQL where clause, and is applied in a population selection pattern."

Primitive: Display Set
[Col. 16, lines 41-44]
"A Display Set is an ordered set of attributes that is shown to reflect the status of an object. A Display Set may be thought of as an abstraction of the columns in a SQL clause, and is applied in a population selection pattern."

Primitive: Order Criterion
[Col. 16, lines 45-50]
"The Order Criterion is an ordered set of tuples that contain: an attribute and an order (ascending/descending). This set of tuples fixes an order criterion over the filtered objects. An order criterion pattern may be thought of as an abstraction of an order by SQL clause, and is applied in a population selection pattern."

I hereby certify that this correspondence is being deposited with the United States Postal Service as First Class Mail, postage prepaid, in an envelope addressed to: Commissioner for Patents , P.O. Box 1450, Alexandria, Va. 22313-1450.
on 2/7/05
(Date of Deposit)

[illegible]